Combinatorial Generation by Fusing Loopless Algorithms

Tadao Takaoka and Stephen Violich

Department of Computer Science and Software Engineering
University of Canterbury
Christchurch, New Zealand
{tad, ssv10}@cosc.canterbury.ac.nz

Abstract. Some combinatorial generation problems can be broken down into subproblems for which loopless algorithms already exist. We discuss means by which existing loopless algorithms for subproblems can be combined or fused to produce a new loopless algorithm that solves the original problem. We demonstrate this method with two new loopless algorithms, MULTPERM and MIXPAR. MULTPERM generates multiset permutations using only arrays and requiring only linear space; it is simpler and more efficient in both time and space than the recent algorithm of Korsh and LaFollette. MIXPAR generates well-formed parenthesis strings containing two different types of parentheses.

1 Introduction

The generation of combinatorial objects, such as combinations, permutations and parenthesis strings, is a well studied area, covered by Nijenhuis and Wilf [6], Reingold, Nievergelt and Deo [7], Wilf [11] and Savage [8].

Loopless algorithms for combinatorial generation were introduced by Ehrlich [2]. These algorithms generate each combinatorial object from its predecessor using no more than a constant number of instructions, thus they are 'loop-free'. It follows that it should be possible to combine loopless algorithms in such a way that the resulting algorithm still satisfies this property. While an arbitrary combination of loopless algorithms would probably not be useful or even meaningful, we can imagine situations in which combining loopless algorithms might be a good approach. If a combinatorial generation problem can be broken down into subproblems for which loopless algorithms already exist, then combining those algorithms might lead to a loopless algorithm for the original problem.

This idea is not new, for example Korsh and Lipschutz [4] and Korsh and LaFollette [5] give loopless algorithms for multiset permutations that combine existing loopless algorithms for element selection and combination movement. We believe, however, that combining loopless algorithms has not been discussed in general before. We refer to the combining of algorithms as *fusing* because this does not limit us to any particular structures or patterns.

We introduce general program structures for fused loopless algorithms and discuss implementation issues in Sect. 2. We cover Williamson's algorithm for variations in Gray code order [12] in Sect. 3, as it is the basis for many of the subsequent algorithms we discuss. We use fusing to produce MULTPERM, a new algorithm for multiset permutations, in Sect. 4, and compare it with a similar algorithm recently published by Korsh and LaFollette [5]. A second new algorithm, MIXPAR, for generating mixed parenthesis strings, is produced by fusing in Sect. 5. Finally, we draw some conclusions in Sect. 6.

2 Fusing Loopless Algorithms

The generalised program structure for a loopless algorithm is shown in Fig. 1(a). Function *init* initialises the algorithm and generates the first object, *next* generates each successive object, while *isnext* returns whether there is another object in the sequence. Functions *next* and *isnext* run in $O(1)$ time, while *init* is allowed $O(n)$ time. The actual loopless algorithm, in the strictest sense, is *next*.

```
1.  init();
2.  do {
3.     next();
4.  } while (isnext());
```

(a) Single algorithm.

```
1.  init();
2.  do {
3.     do {
4.        next1();
5.     } while (isnext1());
6.     next2();
7.     reinit1();
8.  } while (isnext2());
```

(b) Nested algorithms.

```
1.  init();
2.  do {
3.     if (isnext1()) {
4.        next1();
5.     } else {
6.        next2();
7.        reinit1();
8.     }
9.  } while (isnext2());
```

(c) Un-nested algorithms.

Fig. 1. Program structures for loopless algorithms.

Two loopless algorithms can be nested so that a complete cycle of the inner algorithm runs during each iteration of the outer algorithm, as shown in Fig. 1(b). Functions *next1* and *isnext1* belong to the inner algorithm, while *next2* and *isnext2* belong to the outer. Because the initial and final states of a loopless algorithm differ, a new function, *reinit1*, is required to reinitialise the inner algorithm in preparation for each new cycle. There are two ways an algorithm can be reinitialised: *refreshing* means to reset an algorithm to its initial state; *reversing* means to alter the algorithm so it will run from its final state back to its initial state over a cycle. Since reinitialisation occurs between objects, *reinit* is only allowed $O(1)$ time. Although the program for nested loopless algorithms contains an extra do-while loop, successive objects are still generated in no more than a constant number of instructions.

For greater clarity, the nested program structure can be modified into an *un-nested* program structure by replacing the second do-while loop with an if-then-else statement, as shown in Fig. 1(c). This un-nested configuration executes *next1*, *next2* and *reinit1* in exactly the same order as the nested configuration, but now a single, loop-free algorithm that generates exactly one object per iteration can be isolated within the program.

Although *reinit1* is limited to $O(1)$ time, there are a couple of tricks for fitting $O(n)$-time reinitialisation into this framework. For example, the final state of an algorithm might include some array $a_{1\ldots n}$ that has $O(n)$ points of difference from its initial state. Supposing the algorithm is irreversible, then it requires $O(n)$ time to reinitialise. One option, available if the algorithm finishes with different a_i at different stages during its cycle, is to reinitialise each a_i as soon as it becomes obsolete, during iterations of *next1*. In this way, $O(n)$ reinitialising steps can be executed in $O(1)$ time per object, a technique we call *time-stealing*. In the best case, this algorithm would give cues as to exactly when each a_i becomes obsolete; in the worst, a for-loop would be simulated, using a counter variable and an arbitrary start cue. A second option is less elegant and much less efficient, although it seems universally applicable: maintain two separate versions of the troublesome arrays or variables. Then, in any given cycle of the inner algorithm, one version can be used while the other is reinitialised as per time-stealing.

3 Williamson's Algorithm

We include a discussion of Williamson's loopless algorithm [12, p.112] for generating variations in Gray code order because it is a basis for many of the subsequent algorithms presented in this paper. The algorithm generates elements of the product space $S = S_1 \times S_2 \times \ldots \times S_n$, with $S_i = 0, 1, \ldots, r_i - 1$ for $i = 1, 2, \ldots, n$. Williamson's algorithm is shown in Fig. 2.

The variables in Williamson's algorithm are: $v_{1\ldots n}$, the current variation; j, the current position in v to change; $d_{1\ldots n}$, the current increment (1 or -1) for each position in v; and $e_{0\ldots n}$, which determines the order in which positions in v should be selected as values for j.

Values for n and all $r[i]$ are read from the user. The remaining variables are initialised as follows: all v_i are set to 0; all d_i are set to 1; all e_i are set to i; and j is set to n.

```
1.  void next() {
2.      e[n] = n;
3.      v[j] += d[j];
4.      if (v[j] == 0 || v[j] == r[j] − 1)  {e[j] = e[j − 1]; e[j − 1] = j − 1; d[j] = −d[j];
 }
9.      j = e[n];
10. }
```

Fig. 2. Williamson's loopless algorithm for variations in Gray code order.

Array e is used to looplessly simulate a recursive tree traversal. Though this technique is well known and comprises only a few lines of code, it is nontrivial and rarely explained.

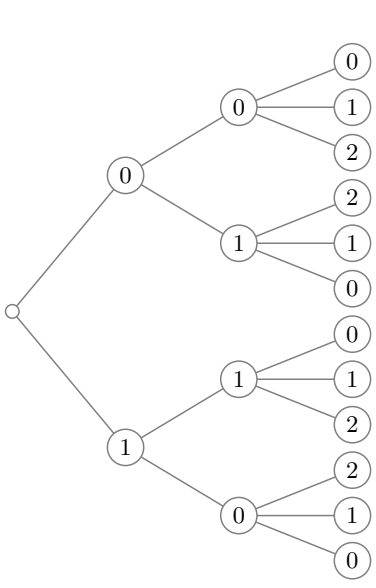
When e_i is set to i, we say that e_i is *reset*, since i was the initialised value of e_i. When v_j becomes a last child, the value at e_{j-1} is passed along one place to e_j, then e_{j-1} is reset. Referring to the coding tree in Fig. 3, this can be seen when $v = 0, 1, 0$. Because v_3 has become a last child, e_3 inherits the value 1 from e_2, while e_2 is reset to 2.

A similar pass-reset pattern occurs between e_n and variable j. At the end of every iteration of *next* the value at e_n is passed along to variable j; at the start of the next iteration, e_n is reset. Referring again to Fig. 3, the resetting of e_3 is visible on every third line beginning with the fourth, when $v = 0, 1, 2$. It *happens* on every line, of course, but can only be seen when e_3 is not changed due to v_3 being a last child, and when e_3 was not already 3.

In effect, e can be thought of as a conveyor belt that passes information along towards variable j. It is helpful to picture variable j as positioned immediately after e_n, since information flows along array e and into j. Whenever information is passed along, the source of that information is reset.

Any value i can only enter the array by resetting e_i. When e_i inherits a value from e_{i-1}, that value instead of i will be carried towards variable j. That means that v_i will be skipped over on the next occasion that would have otherwise been its turn to be changed.

When v_i is skipped, and one of its ancestors is changed, v_i becomes a first child, so it should not be skipped again. Thus, as soon the value of e_i is passed on, e_i is reset. This means the *subsequent*



v	e
0 0 0	0 1 2 3
0 0 1	0 1 2 3
0 0 2	0 1 2 2
0 1 2	0 1 1 3
0 1 1	0 1 1 3
0 1 0	0 1 2 1
1 1 0	0 0 2 3
1 1 1	0 0 2 3
1 1 2	0 0 2 2
1 0 2	0 1 0 3
1 0 1	0 1 0 3
1 0 0	0 1 2 0

Fig. 3. Coding tree, output v (v_1 to v_3) and array e (e_0 to e_3) for Williamson's algorithm for $n = 3$ and $r = 2, 2, 3$.

value to be passed from e_i will be i again, making v_i available for change. Note that if the value of e_i was already i before it was passed along then resetting e_i has no effect.

4 Multiset Permutations

A multiset, or set with repetitions, has k distinct elements, which we assume without loss of generality to be the integers $[1, k]$. Each distinct element i has a multiplicity m_i, which is the number of times it appears in the multiset. The size n of the multiset is the sum of all multiplicities. For example, the multiset $\{1, 1, 1, 2, 2, 3\}$ has $k = 3$, $m = \{3, 2, 1\}$ and $n = 6$. Indistinguishable elements are called *similar*.

Our approach overall is very similar to that of Korsh and LaFollette [5], however our algorithm is simpler and more efficient in both time and space. The approach also has similarities to those of Korsh and Lipschutz [4] and Vajnowszki [10].

Our idea is based on the Johnson and Trotter [3] [9] algorithms for set permutations, which work by recursively moving single elements through subpermutations. We reasoned that a modified algorithm could recursively move groups of similar elements through subpermutations, thereby generating multiset permutations. This grouped element movement could be achieved using a combinations algorithm. The recursive algorithm for multiset permutations is as follows:

Let *perm* be a multiset of n integers. Let $subp_i$ be a subpermutation of *perm* comprising all elements *greater than* i. Initially *perm* is the lexicographically least permutation. If $k = 1$ then *perm* is the only permutation. Otherwise, the 1s are placed among $subp_1$ in all remaining distinct ways such that the relative order of elements of $subp_1$ is maintained, and $subp_1$ is contiguous in the final permutation. This generates all permutations containing $subp_1$. If there is another $subp_1$ of *perm*, it is generated recursively, and the next *perm* becomes this next $subp_1$ bounded by the 1s. The 1s are now placed among this next $subp_1$ in all remaining distinct ways, subject to the same conditions as before. This generates all permutations containing this next $subp_1$. This process of moving 1s through $subp_1$s continues until they have appeared in all distinct ways in the last $subp_1$. When the k integers are distinct this algorithm mimics the Johnson-Trotter.

The recursive algorithm we describe is very similar to that of Korsh and LaFollette, with one important difference: when the similar elements finish moving through a subpermutation, Korsh and LaFollette require that they all be at one end left or right of the subpermutation; we require only that the subpermutation be contiguous, meaning the similar elements may finish distributed across both ends. Our more relaxed requirement meant we had more combinations algorithms to choose from than Korsh and LaFollette. Our multiset permutations algorithm is simpler and more efficient than theirs ultimately because the combinations algorithm we chose was more advantageous.

Thus, our choice of combinations algorithm was guided by the three requirements from our recursive multiset permutations algorithm, plus one guideline: the relative order of 0s must be maintained; the 0s must finish as a contiguous substring; the algorithm must be reversible in O1 time; and transpositions should be limited to O1 distance this avoids significant extra bookkeeping.

The combinations algorithm we chose was that of Chase [1], shown in Fig. 4. We have altered the algorithm so that all decision making is clear optimised shortcuts have been replaced with assumed original conditional statements and so that the algorithm can run both forwards and backwards. Its 1- or 2-apart transpositions means the relative order of both 1s and 0s is easily maintained. It is easily reversible, requiring only the inversion of one boolean function. It starts with 1s *all-left* 1^n0^k and finishes in one of two easily recognisable arrangements: *one-right* $1^{n-1}0^{k-n}1$ iff $n = 1$ or k is even; or *two-right* $1^{n-2}0^{k-n}11$ iff $n > 1$ and k is odd. Another benefit is that it uses very few variables.

The variables in Chase's algorithm are: $comb_{1..n+1}$, the current combination; z, the position in *comb* of the first non-minimal element, that is the lowest i such that $comb_i > i$; and x and y, the values exiting and entering *comb* respectively.

Values for n and r are read from the user. All $comb_i$ are set to i, except $comb_{n+1}$ which is initialised to $2r + 1$. Variable z is set to $n + 1$.

```
1.  void next() {
2.    if (z == 1) {
3.      if (inc(1)) {
4.        if (adj(1)) {if (inc(2)) {move(1, 1, 2); } else {move(2, −1, 2); }          // 1, 2
5.        } else {move(1, 1, 1); }                                                    // 3
6.      } else {move(1, −1, 1); }                                                     // 4
7.    } else {
8.      if (inc(z − 1)) {
9.        if (z > 2 &&inc(z − 2)) {move(z − 2, 1, 2); } else {move(z − 1, 1, 1); }  // 5, 6
10.     } else {
11.       if (!adj(z)) {if (inc(z)) {move(z, 1, 1); } else {move(z, −1, 1); } }     // 7, 8
12.       else {if (inc(z + 1)) {move(z, 1, 2); } else {move(z + 1, −1, 2); }       // 9, 10
13.     }
14.   }
15. }
16. void move(int p, int d, int s) {
17.   x = comb[p]; y = x + d ∗ s;
18.   comb[p] = x + d; comb[p + s − 1 ∗ d] = y;
19.   z+ = s ∗ ((comb[z] == z) − (comb[z − 1]! = z − 1));
20. }
```

Fig. 4. Chase's combinations algorithm. Each iteration results in one of ten possible calls to *move*, numbered in comments on the right.

The functions in Chase's algorithm are: $adj(i)$, which returns whether $comb_i$ and $comb_{i+1}$ are adjacent, that is whether $comb_i + 1 = comb_{i+1}$; and inc(i), which returns whether $comb_i$ is increasing or not, which is equivalent to $comb_{i+1}$ mod 2. In Chase's algorithm, each position's direction is determined by the next position's parity; inverting function $inc()$ makes the algorithm run in reverse.

The many nested if-then-else statements evaluate directions and adjacencies of elements within one or two positions of $comb_z$, the first non-minimal element, to determine what transposition to make. We have isolated this transposition in procedure *move*, whose parameters are the position, direction and span (distance) of the transposition. Further explanation of the logic behind Chase's algorithm is nontrivial and beyond the scope of this paper; we ask the reader to consider the algorithm a black box, and to refer to Chase's paper [1] if detailed understanding is required. Output for Chase's algorithm is shown in Fig. 5.

	comb	bit vector	z	move
1.	1 2 3 4	1 1 1 1 0 0	5	6
2.	1 2 3 5	1 1 1 0 1 0	4	5
3.	1 3 4 5	1 0 1 1 1 0	2	6
4.	2 3 4 5	0 1 1 1 1 0	1	2
5.	1 2 4 5	1 1 0 1 1 0	3	9
6.	1 2 5 6	1 1 0 0 1 1	3	6
7.	1 3 5 6	1 0 1 0 1 1	2	6
8.	2 3 5 6	0 1 1 0 1 1	1	1
9.	3 4 5 6	0 0 1 1 1 1	1	4
10.	2 4 5 6	0 1 0 1 1 1	1	4
11.	1 4 5 6	1 0 0 1 1 1	2	10
12.	1 3 4 6	1 0 1 1 0 1	2	6
13.	2 3 4 6	0 1 1 1 0 1	1	2
14.	1 2 4 6	1 1 0 1 0 1	3	8
15.	1 2 3 6	1 1 1 0 0 1	4	

Fig. 5. Chase's algorithm output for $n = 4$, $r = 6$.

```
1.  void next() {
2.    e[1] = 1;
3.    if (z[j] == 1) {
4.      if (inc(z[j])) {
5.        if (adj(z[j])) {if (inc(z[j] + 1)) {move(z[j], 1, 2); } else {move(z[j] + 1, −1, 2); }
6.        } else {move(z[j], 1, 1); }
7.      } else {move(z[j], −1, 1); }
8.    } else {
9.      if (inc(z[j] − 1)) {
10.       if (z[j] > 2 &&inc(z[j] − 2)) {move(z[j] − 2, 1, 2); } else {move(z[j] − 1, 1, 1); }
11.     } else {
12.       if (!adj(z[j])) {if (inc(z[j])) {move(z[j], 1, 1); } else {move(z[j], −1, 1); } }
13.       else {if (inc(z[j] + 1)) {move(z[j], 1, 2); } else {move(z[j] + 1, −1, 2); } }
14.     }
15.   }
16.   perm[x + o[j]] = perm[y + o[j]]; perm[y + o[j]] = j;
17.   if (a[j] < k) {o[a[j]] = o[a[j]] − b[j] ∗ d[j]; a[j] = a[j] + 1; }
18.   if (comb[j, m[j] − b[j] + 1] == r[j] − b[j] + 1 &&comb[j, m[j] − b[j]] == m[j] − b[j]
19.     ||comb[j, m[j]] == m[j]) {
20.     e[j] = e[j + 1]; e[j + 1] = j + 1; d[j] = −d[j]; a[j] = j + 1;
21.   }
22.   j = e[1];
23. }
```

Fig. 6. MULTPERM, a new multiset permutations algorithm.

To fuse a loopless multiset permutations algorithm from Williamson's and Chase's algorithms required surprisingly few modifications. Each of the k groups of similar elements moves as a combination through its subpermutation, requiring its own Chase data. Thus, Chase's variable z and array $comb_{1\ldots n}$ were extended by one dimension each to $z_{1\ldots k}$ and $comb_{1\ldots k,1\ldots m_i}$ respectively. Each $comb_i$ is of length m_i. Williamson's algorithm was altered to start with $j = 1$ instead of n, and its second (incrementing/decrementing) step was replaced with the modified Chase's algorithm. Thus Williamson's algorithm selects the similar elements j to move, and Chase's algorithm moves them among $subp_j$ in combination fashion, using $comb_j$ and z_j. Algorithm MULTPERM, our new multiset permutations algorithm, is given in Fig. 6; a complete C++ program is given in App. A.

To translate the relative transpositions of elements in Chase combinations to absolute transpositions in the multiset permutation, $perm_{1\ldots n}$, required several new variables: $o_{1\ldots k}$, the absolute offsets for each combination; $a_{1\ldots k}$, which keeps track of the offsets that have been updated for the current j's Chase cycle; and $b_{1\ldots k}$, the number (one or two) of elements that finish right for each combination. For any selected group of similar elements j, each complete Chase cycle displaces subsequent subpermutations by b_j (reverse cycle) or $-b_j$ (forward cycle) positions. Thus all o_i for $i > j$ must be updated during the Chase cycle for j. This is achieved using the time-stealing

1.	1 1 2 2 3	11.	2 3 2 1 1	21.	1 1 3 2 2
2.	1 2 1 2 3	12.	2 3 1 2 1	22.	1 3 1 2 2
3.	2 1 1 2 3	13.	2 1 3 2 1	23.	3 1 1 2 2
4.	2 2 1 1 3	14.	1 2 3 2 1	24.	3 2 1 1 2
5.	2 1 2 1 3	15.	1 2 3 1 2	25.	3 1 2 1 2
6.	1 2 2 1 3	16.	2 1 3 1 2	26.	1 3 2 1 2
7.	1 2 2 3 1	17.	2 3 1 1 2	27.	1 3 2 2 1
8.	2 1 2 3 1	18.	2 1 1 3 2	28.	3 1 2 2 1
9.	2 2 1 3 1	19.	1 2 1 3 2	29.	3 2 1 2 1
10.	2 2 3 1 1	20.	1 1 2 3 2	30.	3 2 2 1 1

Fig. 7. MULTPERM output for $k = 3$, $m = 2, 2, 1$.

	Uniform		Varied	
	KL04	MULTPERM	KL04	MULTPERM
Permutations	168,168,000	168,168,000	75,675,600	75,675,600
Mean Time (s)	31.3	21.5	14.2	9.4

Table 1. Results from experimental evaluation showing that MULTPERM runs 31–34% faster than KL04. Evaluation was over two multisets with many million permutations; multiplicities were uniform $\{3, 3, 3, 3, 3\}$ and varied $\{2, 3, 5, 2, 3\}$ respectively. Both algorithms generated the expected numbers of permutations.

method mentioned in Sec. 2, in which what would be a for-loop is distributed over subsequent calls to function *next*. In this case, over several calls to *next*, a_j counts from $j + 1$ to $k - 1$, and each o_{a_j} is incremented or decremented by b_j. To recognise when forward Chase cycles are complete, that is when combinations are one-right or two-right, array $r_{1\ldots k}$ stores the maximum value that may appear in each of the combinations.

Reversing Chase's algorithm requires no re-initialisation. We have tied function *inc* to Williamson's array d, so changing the sign of d_j inverts *inc*, reversing the algorithm.

MULTPERM runs in constant time per object and requires linear space. Referring to Fig. 6, lines 2, 22–25 and 26 correspond to the first, third and fourth steps of Williamson's algorithm respectively. Lines 3–19 contain Chase's algorithm, while lines 20–21 translate Chase's transpositions to the multiset permutation; these steps together correspond to the second step of Williamson's algorithm. A sample output of MULTPERM for $k = 3$, $m = \{3, 2, 1\}$ is shown in Fig. 7.

We experimentally evaluated MULTPERM against Korsh and LaFollette's algorithm. Both programs were implemented in C++; structure, procedure calls, and I/O were made as similar as possible. Timing included the initialisation and memory-clearing procedures. By convention, output statements were replaced by statements incrementing a counter, whose final value was output to verify that the correct number of objects were generated.

We ran the experiment over two multisets, each with millions of distinct permutations, but with *uniform* and *varied* multiplicities respectively: both multisets had $k = 5$ distinct integers, but *uniform* had $m = \{3, 3, 3, 3, 3\}$ and *varied* had $m = \{2, 3, 5, 2, 3\}$. Our mean times and standard deviation were produced over 10 iterations.

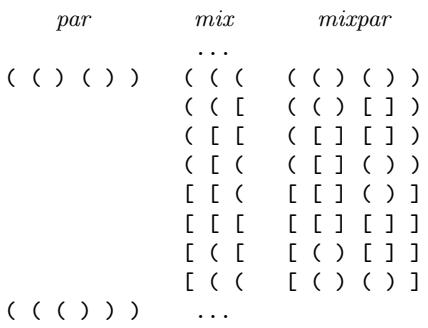
As can be seen from Table 1, MULTPERM runs 31–34% faster than KL04 across both multisets. MULTPERM generated the 168 million permutations of the uniform multiset in an average of 21.5s ($\sigma = 0.11$) to KL04's 31.3s ($\sigma = 0.11$), and the 75 million permutations of the varied multiset in 9.4s ($\sigma = 0.05$) to KL04's 14.2s ($\sigma = 0.05$).

5 Mixed Parenthesis Strings

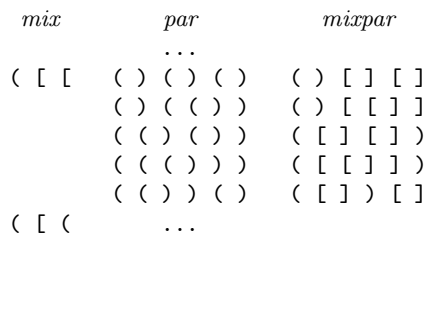
A well-formed parenthesis string, or *par* for short, can be derived from the grammar $P \rightarrow \epsilon \mid (P) \mid PP$. A par has n pairs, and so its size is $2n$.

We introduce a new combinatorial object: *mixed parenthesis strings*, or *mixpars* for short, which comprise parentheses of different types. In this paper we limit the number of types to two, but it is trivial to extend the ideas beyond binary. The grammar for a mixpar is a modification of that for a par, in this case $M \leftarrow \epsilon \mid (M) \mid [M] \mid MM$. Thus, a mixpar is well-formed if its parentheses are arranged as per an ordinary par, *and* if both parentheses in each pair share the same type. For example, `( ) [ ]` and `( [ ] )` are a valid mixpars, while `( ] [ )` and `( [ ) ]` are not.

A mixpar can be thought of as a par with a certain *mix* of types. For example, the mixpar `( ) [ ]` can be described as the par `( ) ( )` with the mix `( [`. Note that with only two types, a mix corresponds to a binary string. It follows that generating all mixpars for some n is a matter of generating either all mixes for each par or all pars for each mix. Thus, an algorithm for generating mixpars nests algorithms for generating pars and mixes in some way. Because loopless algorithms for pars and binary strings exist, we hypothesized that a loopless algorithm for generating mixpars could be fused from these.



(a) par-outside-mix (par-mix).



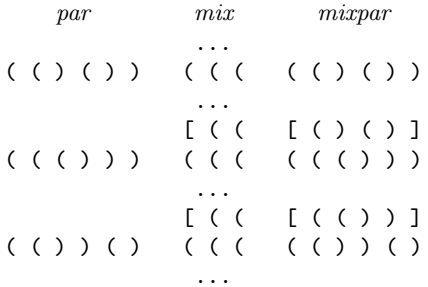
(b) mix-outside-par (mix-par).

Fig. 8. Sample outputs for mixpar algorithms with opposite nesting configurations.

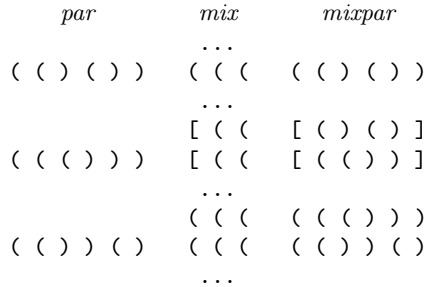
The way in which the two algorithms are nested affects the modifications required to make each algorithm operate directly on mixpars. Figure 8 shows output for mixpar algorithms with each of the possible nesting configurations, par-outside-mix and mix-outside-par. (The par algorithm used is that of Xiang and Ushijima [13], which is the one we ultimately chose and will discuss later; the mix algorithm is simply a Gray code generator.) From Fig. 8(a) it can be seen that for a given iteration of the par algorithm, each iteration of the mix algorithm must change the type of two parentheses (one pair) in the mixpar. Figure 8(b) shows that for a given iteration of the mix algorithm, each iteration of the par algorithm must either swap the places or change the types of two to four parentheses. Thus, the mix-par configuration seemed to require more difficult modification to its inner algorithm, so we opted for the par-mix arrangement.

The method of reinitialising the inner algorithm also has a significant impact on implementing the new algorithm. Figure 9 shows output for mixpar algorithms that refresh and reverse their inner (mix) algorithms respectively. From Fig. 9(a) it can be seen that refreshing the mix algorithm means that all parentheses are round whenever it is the par algorithm's turn to operate. This takes advantage of the fact that the last object in a Gray code has only one point of difference to the first object. Figure 9(b) shows that reversing the mix algorithm means the par algorithm will frequently have to cope with one pair of an alternate type. Again, we opted for the simpler option, being refreshing the mix algorithm and thus avoiding having to make nontrivial modifications to the par algorithm.

In order to change the types of pairs, the mix algorithm needs to know the positions within the mixpar of the parentheses in each pair. Let l_i be the position of the ith left parenthesis, and let r_i be the position of *the partner of* the ith left parenthesis (that is, *n*ot simply the ith right



(a) Refreshing.



(b) Reversing.

Fig. 9. Sample outputs for mixpar algorithms refreshing and reversing the inner mix algorithm respectively.

parenthesis as counted from the start). For example, for the mixpar ((( ))), $l_2 = 2$ and $r_2 = 5$.

Although we do not know of a loopless par algorithm that correctly maintains all l_i and r_i, Xiang and Ushijima's algorithm does correctly maintain all l_i; we now present a method for finding all r_i in constant time per object. We cannot scan the entire mixpar after every iteration of the par algorithm, as that would require $O(n)$ time, so the solution is to use the time-stealing technique mentioned in Section 2, thereby finding each r_i in $O(1)$ time during iterations of the mix algorithm.

We say a parenthesis pair is *empty* if no pairs are nested inside it. Recalling the grammar for a par, the nth pair must be empty, since no subsequent pairs exist. Thus:

$$r_n = l_n + 1 \tag{1}$$

It follows that the $(n-1)$th pair must be empty or nested around the nth pair. Our algorithm is based on the idea that, if we start from the nth pair and work backwards to the first, each pair must be either empty or nested around some substring comprising pairs we have already encountered. Thus, information about substrings must be stored. Let s_{l_i} be the position *a*fter the longest well-formed substring beginning at l_i. For example, for the mixpar ((( ))), $l_2 = 2$ and $s_{l_2} = s_2 = 6$. Because we cannot know all s_i immediately, our algorithm initialises array $s_{1\ldots 2n}$ such that all $s_i = i$; correct s_{l_i} values are set iteratively as will be described. Equation (1) is the base step of our induction. We now show how each successive s_{l_i} and r_i can be found in constant time by working backwards from $i = n$.

If there is no jth left parenthesis immediately after r_i, then the substring beginning at l_i ends at r_i, and s_{r_i+1} will not have changed since initialisation. On the other hand, if r_i is adjacent to some l_j, then the substrings beginning at l_i and l_j end in the same position, and because we are working backwards from the nth pair, s_{l_j} will already have been set correctly. Thus, we derive an unconditional equation that is independent of j:

$$\begin{aligned} s_{l_i} &= \begin{cases} r_i + 1 = s_{r_i+1} \text{ iff } r_i + 1 \neq l_j \\ s_{l_j} \quad\quad = s_{r_i+1} \text{ iff } r_i + 1 = l_j \end{cases} \\ &= s_{r_i+1} \end{aligned} \tag{2}$$

for $1 \leq i \leq n, i < j \leq n$.

The maths for finding r_i is similar. If the $(i+1)$th left parenthesis is not immediately after l_i, then r_i must be, and s_{l_i+1} will not have changed since initialisation. Conversely, if the ith and $(i+1)$th left parentheses are adjacent, then r_i must be immediately after the substring starting at l_{i+1}. Because we are working backwards from the nth pair, $s_{l_{i+1}}$ will already have been set correctly. Thus, we derive another unconditional equation:

$$\begin{aligned} r_i &= \begin{cases} l_i + 1 = s_{l_i+1} \text{ iff } l_i + 1 \neq l_{i+1} \\ s_{l_{i+1}} \quad = s_{l_i+1} \text{ iff } l_i + 1 = l_{i+1} \end{cases} \\ &= s_{l_i+1} \end{aligned} \tag{3}$$

for $1 \leq i \leq n$.

Thus, using (2) and (3), right parentheses from nth to first can be found in $O(1)$ time each, during iterations of the first half of the Gray cycle. As we finish with each r_i during the second half of the Gray cycle, we reset each s_{l_i}.

We now cover Xiang and Ushijima's par algorithm. In addition to correctly maintaining all l_i, it a very efficient loopless par algorithm in terms of time and space. It is also very simple, which helped keep our final MIXPAR algorithm simple. Xiang and Ushijima's algorithm is shown in Fig. 10 (note that we have renamed their array for the positions of the left parentheses from p to l for consistency with our approach).

Xiang and Ushijima's algorithm introduces several new variables. As mentioned, the number of parenthesis pairs is n, which is read from the user. The par is stored in $par_{1\ldots 2n}$, while the left parentheses positions are stored in $l_{1\ldots n}$. These are initialised to ( ) ( ) ... ( ) and 1,3,...,$2n-1$ respectively. Finally, i and c are temporary variables used to facilitate an array swap, storing an

```
1.  void next() {
2.      e[n + 1] = n; i = l[j];
3.      if (d[j] == 1) {if (l[j] == 2 * j − 1) {l[j] = l[j − 1] + 1; } else {l[j] = l[j] + 1; } }
4.      else {if (l[j] == l[j − 1] + 1) {l[j] = 2 * j − 1; } else {l[j] = l[j] − 1; } }
5.      c = par[i]; par[i] = par[l[j]]; par[l[j]] = c;
6.      if (l[j] >= 2 * j − 2) {d[j] = 1 − d[j]; e[j + 1] = e[j]; e[j] = j − 1; }
7.      j = e[n + 1];
8.  }
```

Fig. 10. Xiang and Ushijima's parenthesis strings algorithm.

integer and character respectively. Variables j, $d_{1\ldots n}$ and $e_{0\ldots n}$ are inherited from Williamson's algorithm, and relate to the left parentheses; initialisations remain the same.

It works in the same way as their combinations algorithm from the same paper; both are variations on Williamson's algorithm in which no two elements in the same object can have the same value. Xiang and Ushijima noted that parentheses maintain a relative order, that is $l_1 < l_2 < \ldots < l_n$, and that well-formedness dictates how far to the right each left parenthesis can travel, that is $l_i \leq 2i - 1$ for $1 \leq i \leq n$. In other words, some L_i can travel left until it is adjacent to L_{i-1}, and it can travel right until either it is adjacent to L_{i+1} or it overtakes all R_j where $j < i$. At any time, these principles determine the upper and lower bounds for left parenthesis travel.

Xiang and Ushijima extended Williamson's algorithm to have four patterns of change: O^+, $O^{+\prime}$, O^- and $O^{-\prime}$. The regular positive direction, O^+, causes a parenthesis to move steadily right between its current bounds. The prime positive direction, $O^{+\prime}$, causes a parenthesis to jump from its lower bound to its upper bound, then move steadily left through all remaining values. The negative directions have the opposite effects. These jumps in the prime directions allow the algorithm to avoid clashes (different elements sharing the same value) while generating all combinations of left parenthesis positions.

Output for Xiang and Ushijima's algorithm for $n = 4$ is shown in Fig. 11. All l_i begin maximally, and increment or decrement in a pattern similar, at first glance, to that of Williamson's algorithm. Closer examination of lines 2–5, however, reveals the effect of a prime direction jump. On line 2, l_4 is minimal, so in Williamson's algorithm you would expect it to reverse direction next time it moved. But on line 3, the change to l_3 means that l_4 is no longer minimal. On line 4, a prime jump is employed so that l_4 can take the newly available minimum value before ascending as per usual to the maximum on line 5.

Algorithm MIXPAR, our new mixed parenthesis strings algorithm, is given in Fig. 12. A complete C++ program is given in App. B. Most of the variables in MIXPAR are inherited from

	par	*l*
1.	( ) ( ) ( ) ( )	1 3 5 7
2.	( ) ( ) (( ))	1 3 5 6
3.	( ) (( ) ( ))	1 3 4 6
4.	( ) ((( )))	1 3 4 5
5.	( ) (( )) ( )	1 3 4 7
6.	(( ) ( )) ( )	1 2 4 7
7.	(( ) (( )))	1 2 4 5
8.	(( ) ( ) ( ))	1 2 4 6
9.	((( )) ( ))	1 2 3 6
10.	((( ) ( )))	1 2 3 5
11.	(((( ))))	1 2 3 4
12.	((( ))) ( )	1 2 3 7
13.	(( )) ( ) ( )	1 2 5 7
14.	(( )) (( ))	1 2 5 6

Fig. 11. Xiang and Ushijima's algorithm output for $n = 4$.

```
1.  void next() {
3.    if (jj > 0) {
5.      ee[n + 1] = n;
6.      if (dd[1] > 0 &&jj == t) {
7.        r[jj] = s[l[jj] + 1];
8.        if (jj > 1) {s[l[jj]] = s[r[jj] + 1]; t = t − 1; }
9.      }
10.     if (par[l[jj]] ==′ (′) {par[l[jj]] =′ [′; par[r[jj]] =′]′; }
11.     else {par[l[jj]] =′ (′; par[r[jj]] =′)′; }
12.     ee[jj + 1] = ee[jj]; ee[jj] = jj − 1; dd[jj] = −dd[jj];
13.     if (dd[1] < 0 &&jj == t) {s[l[jj]] = l[jj]; t = t + 1; }
14.     jj = ee[n + 1];
15.   } else {
17.     par[l[1]] =′ (′; par[r[1]] =′)′;
18.     jj = n; t = n;
19.     dd[1] = 1; ee[n] = n − 1;
21.     e[n + 1] = n; i = l[j];
22.     if (d[j] > 0) {if (l[j] == 2 ∗ j − 1) {l[j] = l[j − 1] + 1; } else {l[j] = l[j] + 1; } }
23.     else {if (l[j] == l[j − 1] + 1) {l[j] = 2 ∗ j − 1; } else {l[j] = l[j] − 1; } }
24.     c = par[i]; par[i] = par[l[j]]; par[l[j]] = c;
25.     if (l[j] > 2 ∗ j − 3) {e[j + 1] = e[j]; e[j] = j − 1; d[j] = −d[j]; }
26.     j = e[n + 1];
27.   }
28. }
```

Fig. 12. MIXPAR, a mixed parenthesis strings algorithm.

its constituent algorithms. From Xiang and Ushijima's algorithm come the variables n, $par_{1\ldots 2n}$, $l_{1\ldots n}$, j, $d_{1\ldots n}$, $e_{0\ldots n}$, i and c. From Williamson's algorithm, to run our mix (Gray code) algorithm, come the variables jj, $dd_{1\ldots n}$ and $ee_{0\ldots n}$. All initialisations are as previously described.

Three new variables are introduced. Finding right parentheses requires arrays $r_{1\ldots n}$ and $s_{1\ldots 2n}$, of which r is not initialised and the initialisation of s has already been covered. Finally, to keep track of which right parenthesis is due to be found during the first half of the Gray cycle, and which value of s is due to be refreshed during the second half, we use variable t; initially $t = n$.

A sample output of MIXPAR for $n = 3$ is shown in Fig. 13. The output is displayed in columns, where each column begins with a par generated by Xiang and Ushijima's algorithm (lines 1, 9, 17, 25 and 33). The remaining lines in each column show complete Gray code cycles of mixes for that column's par.

6 Conclusions

We have taken the idea of developing new loopless algorithms by combining or fusing existing loopless algorithms, and derived some generally applicable theory. We applied our theory to two combinatorial generation problems: multiset permutations and mixed parenthesis strings. Our

```
1. ( ) ( ) ( )    9. ( ) ( ( ) )   17. ( ( ) ( ) )   25. ( ( ( ) ) )   33. ( ( ) ) ( )
2. ( ) ( ) [ ]   10. ( ) ( [ ] )   18. ( ( ) [ ] )   26. ( ( [ ] ) )   34. ( ( ) ) [ ]
3. ( ) [ ] [ ]   11. ( ) [ [ ] ]   19. ( [ ] [ ] )   27. ( [ [ ] ] )   35. ( [ ] ) [ ]
4. ( ) [ ] ( )   12. ( ) [ ( ) ]   20. ( [ ] ( ) )   28. ( [ ( ) ] )   36. ( [ ] ) ( )
5. [ ] [ ] ( )   13. [ ] [ ( ) ]   21. [ [ ] ( ) ]   29. [ [ ( ) ] ]   37. [ [ ] ] ( )
6. [ ] [ ] [ ]   14. [ ] [ [ ] ]   22. [ [ ] [ ] ]   30. [ [ [ ] ] ]   38. [ [ ] ] [ ]
7. [ ] ( ) [ ]   15. [ ] ( [ ] )   23. [ ( ) [ ] ]   31. [ ( [ ] ) ]   39. [ ( ) ] [ ]
8. [ ] ( ) ( )   16. [ ] ( ( ) )   24. [ ( ) ( ) ]   32. [ ( ( ) ) ]   40. [ ( ) ] ( )
```

Fig. 13. MIXPAR algorithm output for $n = 3$.

multiset permutations algorithm achieves linear space, surpassing the recent algorithm of Korsh and LaFollette in efficiency and elegance. Our mixed parenthesis string algorithm achieves constant time per object and linear space, the first algorithm to do so.

We hope that our theory will be of use to others attempting to combine loopless algorithms, and that this approach offers another means to tackle loopless combinatorial generation problems.

References

1. P. J. Chase. Combination generation and graylex ordering. *Congressus Numerantium*, 69:215–242, 1989.
2. Gideon Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. ACM*, 20(3):500–513, 1973.
3. S. M. Johnson. The generation of permutations by adjacent transpositions. *Math. Comp.*, 17:282–285, 1963.
4. James Korsh and Seymour Lipschutz. Generating multiset permutations in constant time. *J. Alg.*, 25(2):321–335, 1997.
5. James F. Korsh and Paul S. LaFollette. Loopless array generation of multiset permutations. *The Computer Journal*, 47(5):612–621, 2004.
6. A. Nijenhuis and H. S. Wilf. *Combinatorial Algorithms*. Academic Press, 1975.
7. E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, 1977.
8. Carla Savage. A survey of combinatorial gray codes. *SIAM Review*, 39(4):605–629, 1997.
9. H. F. Trotter. Perm (algorithm 115). *Commun. ACM*, 5(8):434–435, 1962.
10. Vincent Vajnovszki. A loopless algorithm for generating the permutations of a multiset. *Theor. Comput. Sci.*, 307(2):415–431, 2003.
11. H. S. Wilf. *Combinatorial Algorithms: An Update*. SIAM, 1989.
12. S. Gill Williamson. *Combinatorics for Computer Science*. Computer Science Press, 1985.
13. Limin Xiang and Kazuo Ushijima. On o(1) time algorithms for combinatorial generation. *Comput. J.*, 44(4):292–302, 2001.

A MULTPERM Pascal Program

```
program multperm;
const MAXK = 9; MAXM = 9; MAXN = MAXK*MAXM;
var k, n, j, x, y : integer;
    perm : array [1..MAXN] of integer;
    comb : array [1..MAXK,1..MAXM+1] of integer;
    m, d, o, r, z, a, b : array [1..MAXK] of integer;
    e : array [0..MAXK] of integer;

function adj(i : integer) : boolean; begin adj := comb[j,i]+1 = comb[j,i+1]; end;

function inc(i : integer) : boolean; begin inc := comb[j,i+1] mod 2 = integer(d[j]>0); end;

procedure input; var i : integer; begin read(k); for i := 1 to k do read(m[i]); end;

procedure output; var i : integer; begin for i := 1 to n do write(perm[i]:2); writeln; end;

procedure init;
var i : integer;
begin
        o[1] := 0;
        for i := 2 to k do o[i] := o[i-1]+m[i-1];
        r[k] := m[k];
        for i := k-1 downto 1 do r[i] := r[i+1]+m[i];
        n := o[k]+m[k];
        for i := 1 to k do for j := 1 to m[i] do perm[j+o[i]] := i;
        for i := 1 to k do d[i] := 1;
        for i := 1 to k+1 do e[i] := i;
        for i := 1 to k do begin
        for j := 1 to m[i] do comb[i,j] := j;
        comb[i,m[i]+1] := 2*r[i]+1;
        z[i] := m[i]+1;
        end;
        for i := 1 to k-1 do a[i] := i+1;
        for i := 1 to k-1 do b[i] := 1+integer((m[i]>1) and (r[i] mod 2=1));
    j := 1;
end;

procedure next;
var i, u, v, w : integer;
begin
    e[1] := 1;
    {Next comb, perm by Chase}
    if z[j]=1 then begin
        v := 1;
            if inc(1) then begin
            if adj(1) then begin u := 2; w := 2*integer(inc(2))-1; end
                      else begin u := 1; w := 1; end;
        end else begin u := 1; w := -1; end;
    end else begin
        if inc(z[j]-1) then begin
            u := integer((z[j]>2) and inc(z[j]-2))+1; v := z[j]-u; w := 1;
        end else begin
            v := z[j]; u := 1+integer(adj(v)); w := 2*integer(inc(v-1+u))-1;
        end;
    end;
    i := v+(w-1)*(u-1) div -2;
    x := comb[j,i]; y := x+u*w;
    comb[j,i] := x+w; comb[j,i+(u-1)*w] := y;
    z[j] := z[j]-integer(comb[j,v]=v)*u*w-integer(v<z[j])*u;
    perm[x+o[j]] := perm[y+o[j]]; perm[y+o[j]] := j;
    if a[j]<k then begin o[a[j]] := o[a[j]]-b[j]*d[j]; a[j] := a[j]+1; end;
    {Next j by Johnson-Trotter, re-initialise Chase}
    if (comb[j,m[j]-b[j]+1]=r[j]-b[j]+1) and (comb[j,m[j]-b[j]]=m[j]-b[j])
    or (comb[j,m[j]]=m[j]) then begin
        d[j] := -d[j]; e[j] := e[j+1]; e[j+1] := j+1; a[j] := j+1;
    end;
    j := e[1];
end;

begin
   input;
   init;
   output;
   repeat
      next;
      output;
   until j = k;
end.
```

B MIXPAR Pascal Program

```
program mixpar;
const MAXN = 100;
var par : array[1..2*MAXN] of char;
    n, j, jj, t : integer;
    d, dd, l, r : array[1..MAXN] of integer;
    e, ee : array[1..MAXN+1] of integer;
    s : array[1..2*MAXN] of integer;

procedure input; begin read(n); end;

procedure output;
var i : integer;
begin
    for i := 1 to 2*n do write(par[i], ' '); writeln;
end;

procedure init;
var i : integer;
begin
    for i := 1 to n do begin par[2*i-1] := '('; par[2*i] := ')'; end;
    j := n;
    for i := 1 to n do d[i] := 1;
    for i := 1 to n+1 do e[i] := i-1;
    jj := n;
    for i := 1 to n do dd[i] := 1;
    for i := 1 to n+1 do ee[i] := i-1;
    for i := 1 to n do l[i] := 2*i-1;
    for i := 1 to 2*n do s[i] := i;
    t := n;
end;

procedure next;
var
    i : integer;
    c : char;
begin
    if jj > 0 then begin
        {next_gray}
        ee[n+1] := n;
        if (dd[1] > 0) and (jj = t) then begin
            r[jj] := s[l[jj]+1];
            if jj > 1 then begin s[l[jj]] := s[r[jj]+1]; t := t-1; end;
        end;
        if par[l[jj]] = '(' then begin par[l[jj]] := '['; par[r[jj]] := ']'; end
                            else begin par[l[jj]] := '('; par[r[jj]] := ')'; end;
        ee[jj+1] := ee[jj]; ee[jj] := jj-1; dd[jj] := -dd[jj];
        if (dd[1] < 0) and (jj = t) then begin s[l[jj]] := l[jj]; t := t+1; end;
        jj := ee[n+1];
    end else begin
        {reinit_gray}
        par[l[1]] := '('; par[r[1]] := ')';
        jj := n; t := n;
        dd[1] := 1; ee[n] := n-1;
        {next_xupar}
        e[n+1] := n; i := l[j];
        if d[j] > 0 then begin if l[j] = 2*j-1 then l[j] := l[j-1]+1 else l[j] := l[j]+1; end
                    else begin if l[j] = l[j-1]+1 then l[j] := 2*j-1 else l[j] := l[j]-1; end;
        c := par[i]; par[i] := par[l[j]]; par[l[j]] := c;
        if l[j] > 2*j-3 then begin e[j+1] := e[j]; e[j] := j-1; d[j] := -d[j]; end;
        j := e[n+1];
    end;
end;

begin
    input;
    init;
    output;
    repeat
        next;
        output;
    until (j = 1) and (jj = 0);
end.
```